mdalayzer: A Molecular Dynamics Analysis Package

November 21, 2014

Group Members:
Mike Howard (`mphoward@princeton.edu`)
Sang Beom Kim (`sangbk@princeton.edu`)
Andrew Santos (`apsantos@princeton.edu`)
Elia Altabet (`altabet@princeton.edu`)
Joey Vella (`vella@princeton.edu`)

1 Overview

We will create a package that can be used to analyze trajectories from molecular dynamics simulations. Molecular dynamics is a simulation method where Newton's equations of motions are solved for a set of particles in a given interatomic potential. Molecular dynamics has been extremely successful in simulating a wide range of materials and phenomena, and serves as a powerful complement to experimental studies. Various structural and thermodynamic properties of interest can be derived from the molecular trajectory both on the microscopic and macroscopic scale. These properties are usually calculated by post-processing simulation data.

Currently, a variety of well-developed packages exist for performing molecular dynamics simulation on multiple CPUs or GPUs, including GROMACS, LAMMPS, and HOOMD-blue. However, each of these simulation packages outputs trajectories in a variety of formats, including individual ASCII snapshot frames (.xyz, .xml, .gro, .pdb, etc.) and complete binary trajectories (.dcd, .xtc, .trr, etc.). Some simulations are conducted using in-house or legacy codes, which may output frames in customized or archaic tabular formats.

The vast array of available trajectory file formats considerably complicates writing code to perform post-simulation analysis. This often leads to duplication of coding efforts to perform simple tasks, or necessitates interconversion between file formats. Our software, mdalyzer, will vastly simplify this process through a highly extensible Python/C++ module that allows for reading of a variety of typical or customized trajectories that may be analyzed using a standard toolbox.

This toolbox will include methods to calculate the mean-square displacement, the velocity autocorrelation function, density and temperature profiles, the pair distribution function, and the static structure factor. We will also be able to identify the presence of clusters. This toolbox can be easily extended by the user to include other compute methods through the C++ API.

We note that there currently exist some tools for reading and converting molecular dynamics trajectories. However, typically analysis scripts still need to be written in these programs. Our software improves on these codes in three important ways: (1) less work is required for the user conducting standard calculations, (2) C++ computes may calculate on large trajectories faster than scripted computes, and (3) our software is easily extended in C++ for the user reading a new file format or performing a new compute.

2 Architecture

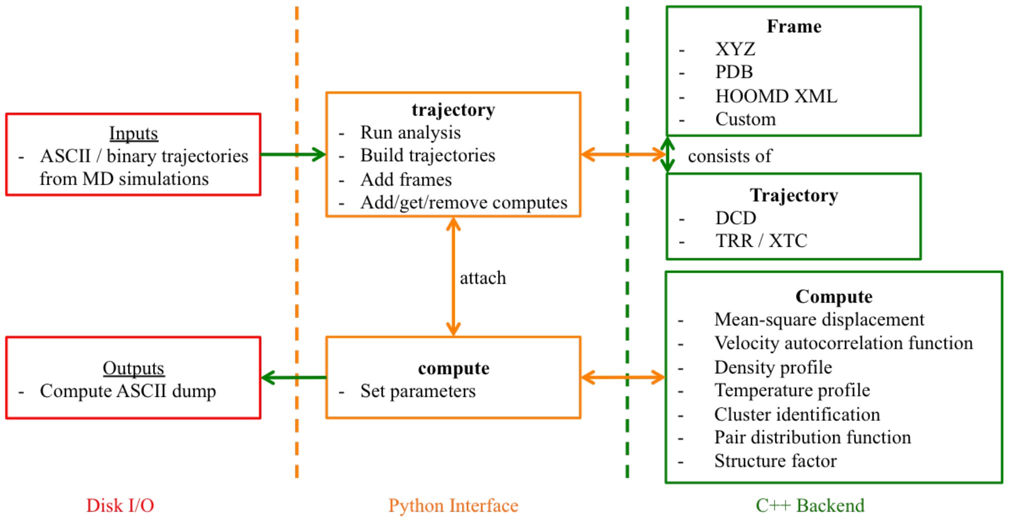

Figure 1: Diagram of software architecture.

2.1 Basic organization

The bulk of our code will be written in C++ with a Python wrapper interface (see diagram in Figure 1). We choose to perform all calculations in C++ because in general reading frames and performing compute loops will be much faster in C++ than Python.

Our program will be organized around a central `Trajectory` object. An array of `Frame` objects is attached to `Trajectory`. Each `Frame` is a snapshot of the simulation at a certain time, and contains information about particle positions, velocities, types, masses, and diameters. We then attach `Compute` objects to the `Trajectory`. Each `Compute` performs some type of analysis on the `Trajectory`, such as calculating the average density profile, the mean-square displacement of atoms, etc. After initialization and attachment of all necessary `Frame` and `Compute` objects, the `Trajectory` has a driving `analyze()` method which (1) reads all `Frame` objects into memory, and (2) calls all attached `Compute` objects sequentially on the `Trajectory`. Each `Compute` dumps its output to disk as it finishes.

2.2 User Interface

For the typical user, all calculations will be interfaced through a Python module. The Python module will allow the user to:

1. Construct a `Trajectory` via a simple initializer. For binary (complete) trajectories, the user need only specify the file name.

   ```
   traj = trajectory.dcd(start_file='start.xml', binary_file='trajectory.dcd')
   ```

   For frame-based trajectories, a simple Python wrapper for attaching frames will be included:

   ```
   traj = trajectory.hoomd()
   traj.addFrame(['dump.0.xml','dump.1.xml'])
   traj.addFrame('dump.3.xml')
   ```

2. Attach multiple `Compute`s via a simple initializer.

   ```
   traj = trajectory.dcd(...)
   traj_2 = trajectory.hoomd(...)

   my_compute = compute.msd(file_name='msd.dat', period=10.0)
   traj.attach(my_compute)

   my_other_compute = compute.density(file_name='density.dat', dx=0.1, dz=0.5)
   traj_2.attach([my_compute,my_other_compute])
   ```

3. Initialize trajectory analysis, which writes output to disk.

   ```
   trajectory.analyze()
   ```

For the more advanced user, all important member variables and functions are exposed in the C++ classes directly. This provides a direct API for users to extend the current library with their own C++ code.

2.3 External libraries

We will capitalize on several existing external packages to simplify some tasks:

1. **Boost** (`www.boost.org`). We will use the Boost libraries for exporting to Python and to perform unit tests on the C++ code. Boost is a standard library on many high performance computing clusters, and also allows us to use smart pointers without requiring C++11 standards. Our code should then be backwards compatible for older operating systems and compilers, as is often the case for clusters maintained individually by groups rather than the university.

2. **pugixml** (`www.pugixml.org`). We will use the pugixml library to parse the HOOMD XML file format. pugixml is light-weight and only requires the compilation of a single file at runtime.

3. **NAMD DCD reader** (`http://www.ks.uiuc.edu/Research/namd/`). We will use the open sourced code from NAMD for reading DCD binary files. This is important because DCD files often have non-standard formatting.

3 Scenarios

Scenario 1: Student or average researcher in molecular simulation field

Molecular simulation programs generally have the capability to return information about the molecular trajectories to the user. The user performs structural and thermodynamic analysis on the results from the most popular molecular simulation packages, in-house, or legacy programs. The user easily analyzes the data from a simple Python script with built-in readers and computes.

Scenario 2: Researcher using multiple simulation programs

Users acquiring output from different simulation packages are often required to convert all files into a single format for post-processing. The user reads frames from multiple file types, and easily performs post-processing with only simple modifications in a single Python script.

Scenario 3: Researcher performing advanced analysis

The user may access the C++ API to extend the current code base, either as a plugin for the user for a specific application or as a contribution to the main code base for more broadly useful routines. For example, if the user wishes to calculate Maxwell-Stefan transport coefficients for mixtures they can extend the mean-squared displacement and/or velocity autocorrelation compute(s).

4 Milestones

Prototype (5 Dec 2014)

- **Base classes / C++ API.** Base classes for `Trajectory`, `Frame`, and `Compute`, to be extended for specific implementations. Headed by Mike Howard.
- **Export to Python**. Boost Python module export of C++ classes. Allows for quick testing of interface. Headed by Sang Beom Kim and Elia Altabet.
- **HOOMD XML.** Representative implementation of a `Frame`. Necessary to test `Compute` on real data. Headed by Andrew Santos.
- **Density profile compute.** Representative implementation of a `Compute`. Needed to validate `Trajectory.analyze()` loop is functional. Headed by Joey Vella.

Alpha version (12 Dec 2014)

- **Additional readers completed.** XYZ file format and PDB file format. Allows for testing of trajectories compiled from different frames. Headed by Joey Vella and Sang Beom Kim.
- **Additional computes completed.** Mean-square displacement and pair distribution function. Headed by Andrew Santos and Elia Altabet.
- **Python module written.** Python classes and helper functions for user interface to wrap C++ methods. Headed by Mike Howard.

Beta version (6 Jan 2015)

- **Fully implement proposed reader formats.** Includes additional ASCII and binary file formats. Headed by Andrew Santos and Sang Beom Kim.
- **Fully implement proposed computes.** Headed by Joey Vella and Elia Altabet.
- **Finalize Python interface.** Complete Python module for all additional classes, and refine user interface. Headed by Mike Howard.

5 Future Features

We considered additional features for our project that we believe to currently be beyond the scope of what can be accomplished. However, we will look to incorporate these features into our software in the future for use in our research groups. Planned future features include:

- **Interconversion of file formats.** (target v. 0.1). Given that trajectories are currently read into memory as a simple structure, we can add the ability to now *export* this data to allow for convenient interconversion. This may be especially useful if parsing an ASCII format into a binary format.

- **Topologies.** (target v. 0.2). The initial version of our project will not consider the calculation of properties that rely on knowledge of the bond structure (topology) of the system being studied. Additional computes could be added for topology-dependent properties. For example if one is interested in gyration tensor of a system of polymers, information on how monomers are bonded would be required. This would also require the augmentation of readers in order to read in topology information.

- **OpenMPI.** (target v. 0.3). Many of the computed properties of trajectories have algorithms that are trivially parallelizable, and so a parallel implementation of the C++ code presents an opportunity for considerable computational speed-up. Parallelization across multiple nodes has the potential to significantly reduce memory requirements because the entire trajectory may not be needed for each parallel rank. An OpenMPI implementation of our code is then an attractive option for performing calculations on large trajectories where computational time and memory are limiting.